ZKPDL: A Language-Based System for Efficient Zero-Knowledge Proofs and Electronic Cash

Sarah Meiklejohn
University of California, San Diego
smeiklej@cs.ucsd.edu

C. Chris Erway
Brown University
cce@cs.brown.edu

Alptekin Küpçü
Brown University
kupcu@cs.brown.edu

Theodora Hinkle
University of Wisconsin, Madison
thea@cs.wisc.edu

Anna Lysyanskaya
Brown University
anna@cs.brown.edu

Abstract

In recent years, many advances have been made in cryptography, as well as in the performance of communication networks and processors. As a result, many advanced cryptographic protocols are now efficient enough to be considered practical, yet research in the area remains largely theoretical and little work has been done to use these protocols in practice, despite a wealth of potential applications.

This paper introduces a simple description language, *ZKPDL*, and an interpreter for this language. ZKPDL implements non-interactive zero-knowledge proofs of knowledge, a primitive which has received much attention in recent years. Using our language, a single program may specify the computation required by both the prover and verifier of a zero-knowledge protocol, while our interpreter performs a number of optimizations to lower both computational and space overhead.

Our motivating application for ZKPDL has been the efficient implementation of electronic cash. As such, we have used our language to develop a cryptographic library, *Cashlib*, that provides an interface for using e-cash and fair exchange protocols without requiring expert knowledge from the programmer.

1 Introduction

Modern cryptographic protocols are complicated, computationally intensive, and, given their security requirements, require great care to implement. However, one cannot expect all good cryptographers to be good programmers, or vice versa. As a result, many newly proposed protocols—often described as efficient enough for deployment by their authors—are left unimplemented, despite the potentially useful primitives they offer to system designers. We believe that a lack of high-level software support (such as that provided by OpenSSL, which provides basic encryption and hashing) presents a barrier to the implementation and deployment of advanced cryptographic protocols, and in this work attempt to remove this obstacle.

One particular area of recent cryptographic research which has applications for privacy-preserving systems is *zero-knowledge proofs* [47, 46, 17, 39], which provide a way of proving that a statement is true without revealing anything beyond the validity of the statement. Among the applications of zero-knowledge proofs are electronic voting [49, 56, 38, 51], anonymous authentication [21, 36, 62], anonymous electronic ticketing for public transportation [50], verifiable outsourced computation [9, 43], and essentially any system in which honesty needs to be enforced without sacrificing privacy. Much recent attention has been paid to protocols based on anonymous credentials [30, 35, 24, 26, 11, 8], which allow users to anonymously prove possession of a valid credential (e.g., a driver's license), or prove relationships based on data associated with that credential (e.g., that a user's age lies within a certain range) without revealing their identity or other data. These protocols also prevent the person verifying a credential and the credential's issuer from colluding to link activity to specific users. As corporations and governments move to put an increasing amount of personal information online, the need for efficient privacy-preserving systems has become increasingly important and a major focus of recent research.

Another application of zero-knowledge proofs is electronic cash. The primary aim of our work has been to enable the efficient deployment of secure, anonymous electronic cash (e-cash) in network applications. Like physical coins, e-coins cannot be forged; furthermore, given two e-coins it is impossible to tell who spent them, or even if they came from the same user. For this reason, e-cash holds promise for use in anonymous settings and privacy-preserving applications, where free-riding by users may threaten a system's stability.

Actions in any e-cash system can be characterized as in Figure 1. There are two centralized entities: the bank and the arbiter. The bank keeps track of users' ac-

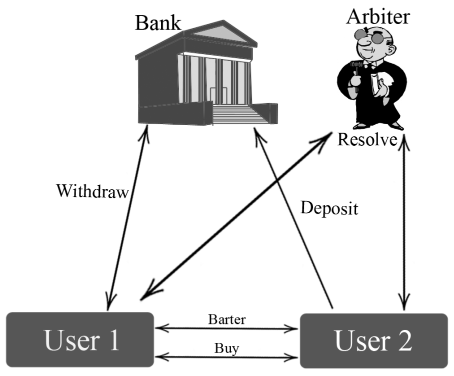

Figure 1: An overview of the entities involved in our e-cash system. Users may engage in buy or barter transactions, withdraw and deposit coins as necessary, and consult the arbiter for resolution only in the case of a dispute.

count balances, lets the users withdraw money, and accepts coin deposits. The arbiter (a trusted third party) resolves any disputes that arise between users in the course of their fair exchanges. Once the users have obtained money from the bank, they are free to exchange coins for items (or just barter for items) and in this way create an economy.

In previous work [10] we describe a privacy-preserving P2P system based on BitTorrent that uses our e-cash and fair exchange protocols to incentivize users to share data. Here, the application of e-cash provides protection against selfish peers, as well as an incentive to upload for peers who have completed their download and thus have no need to continue participating. This system has been realized by our work on the Buy and Barter protocols, described in Section 6.2, which allow a user to fairly exchange e-coins for blocks of data, or barter one block of data for another.

These e-cash protocols can also be used for payments in other systems that face free-riding problems, such as anonymous onion routing [27]. In such a system, routers would be paid for forwarding messages using e-cash, thus providing incentives to route traffic on behalf of others in a manner similar to that proposed by Androulaki et al. [2]. Since P2P systems like these require each user to perform many cryptographic exchanges, the need to provide high performance for repeated executions of these protocols is paramount.

1.1 Our contribution

In this paper, we hope to bridge the gap between design and deployment by providing a language, ZKPDL (Zero-Knowledge Proof Description Language), that enables programmers and cryptographers to more easily implement privacy-preserving protocols. We also provide a library, Cashlib, that builds upon our language to provide simple access to cryptographic protocols such as electronic cash, blind signatures, verifiable encryption, and fair exchange.

The design and implementation of our language and library were motivated by collaborations with systems researchers interested in employing e-cash in high-throughput applications, such as the P2P systems described earlier. The resulting performance concerns, and the complexity of the protocols required, motivated our library's focus on performance and ease of use for both the cryptographers designing the protocols and the systems programmers charged with putting them into practice. These twin concerns led to our language-based approach and work on the interpreter.

The high-level nature of our language brings two benefits. First, it frees the programmer from having to worry about the implementation of cryptographic primitives, efficient mathematical operations, generating and processing messages, etc.; instead, ZKPDL allows the specification of a protocol in a manner similar to that of theoretical descriptions. Second, it allows our library to make performance optimizations based on analysis of the protocol description itself.

ZKPDL permits the specification of many widely-used zero-knowledge proofs. We also provide an interpreter that generates and verifies proofs for protocols described by our language. The interpreter performs optimizations such as precomputation of expected exponentiations, translations to prevent redundant proofs, and caching compiled versions of programs to be loaded when they are used again on different inputs. More details on these optimizations are provided in Section 4.2.

Our e-cash library, Cashlib, described in Section 6, sits atop our language to provide simple access to higher-level cryptographic primitives such as e-cash [27], blind signatures [25], verifiable encryption [28], and optimistic fair exchange [10, 52]. Because of the modular nature of our language, we believe that the set of primitives provided by our library can be easily extended to include other zero-knowledge protocols.

Finally, we hope that our efforts will encourage programmers to use (and extend) our library to implement their cryptographic protocols, and that our language will make their job easier; we welcome contribution by our fellow researchers in this effort. Documentation and source code for our library can be found online at `http://github.com/brownie/cashlib`.

2 Cryptographic Background

There are two main modern cryptographic primitives used in our framework: *commitment schemes* and *zero-*

knowledge proofs. Briefly, a commitment scheme can be thought of as cryptographically analogous to an envelope. When a user Alice wants to commit to a value, she puts the value in the envelope and seals it. Upon receiving a commitment, a second user Bob cannot tell which value is in the envelope; this property is called *hiding* (in this analogy, let's assume Alice is the only one who can open the envelope). Furthermore, because the envelope is sealed, Alice cannot sneak another value into the envelope without Bob knowing: this property is called *binding*. To eventually reveal the value inside the envelope, all Alice has to do is open it (cryptographically, she does this by revealing the private value and any randomness used to form the commitment; this collection of values is aptly referred to as the *opening* of the commitment). We employ both Pedersen commitments [65] and Fujisaki-Okamoto commitments [42, 37], which rely on the security of the Discrete Log assumption and the Strong RSA assumption respectively.

Zero-knowledge proofs [47, 46] provide a way of proving that a statement is true to someone without that person learning anything beyond the validity of the statement. For example, if the statement were "I have access to this sytem" then the verifier would learn only that I really do have access, and not, for example, how I gain access or what my access code is. In our library, we make use of sigma proofs [34], which are three-message proofs that achieve a weaker variant of zero-knowledge known as *honest-verifier zero-knowledge*. We do not implement sigma protocols directly; instead, we use the Fiat-Shamir heuristic [41] that transforms sigma protocols into non-interactive (fully) zero-knowledge proofs, secure in the random oracle model [13].

A primitive similar to zero-knowledge is the idea of a *proof of knowledge* [12], in which the prover not only proves that a statement is true, but also proves that it knows a reason why the statement is true. Extending the above example, this would be equivalent to proving the statement "I have access to the system, and I know a password that makes this true."

In addition to these cryptographic primitives, our library also makes uses of hash functions (both universal one-way hashes [61] and Merkle hashes [60]), digital signatures [48], pseudo-random functions [45], and symmetric encryption [33]. The security of the protocols in our library relies on the security of each of these individual components, as well as the security of any commitment schemes or zero-knowledge proofs used.

3 Design

The design of our library and language arose from our initial goal of providing a high-performance implementation of protocols for e-cash and fair exchange for use in applications such as those described in the introduction. For these applications, the need to support many repeated interactions of the same protocol efficiently is a paramount concern for both the bank and the users. In the bank's case, it must conduct withdraw and deposit protocols with every user in the system, while in the user's case it is possible that a user would want to conduct many transactions using the same system parameters.

Motivated by these performance requirements, we initially developed a more straightforward implementation of our protocols using C++ and GMP [44], but found that our ability to modify and optimize our implementation was hampered by the complexity of our protocols. High-level changes to protocols required significant effort to re-implement; meanwhile, potentially useful performance optimizations became difficult to implement, and there was no way to easily extend the functionality of the library.

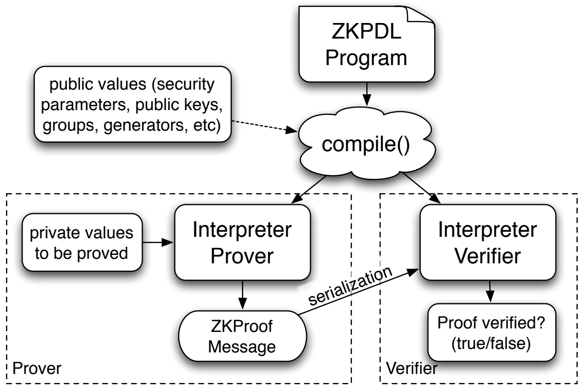
ZKPDL Program
public values (security parameters, public keys, groups, generators, etc)
compile()
private values to be proved
Interpreter Prover
Interpreter Verifier
serialization
ZKProof Message
Proof verified? (true/false)
Prover
Verifier

Figure 2: Usage of a ZKPDL program: the same program is compiled separately by the prover and verifier, who may also be provided with a set of fixed public parameters. This produces an Interpreter object, which can be used by the prover to prove to a verifier that his private values satisfy a certain set of relationships. Serialization and processing of proof messages are provided by the library. Once compiled, an interpreter can be re-used on different private inputs, using the same public parameters that were originally provided.

These difficulties led to our current design, illustrated in Figure 2. Our system allows a pseudocode-like description of a protocol to be developed using our description language, ZKPDL. This program is compiled by our interpreter, and optionally provided a list of public parameters, which are "compiled in" to the program. At compile time, a number of transformations and optimizations are performed on the abstract syntax tree produced by our parser, which we developed using the ANTLR parser generator [64]. Once compiled, these interpreter objects can be used repeatedly by the prover to generate zero-knowledge proofs about private values, or by the verifier to verify these proofs.

Key to our approach is the simplicity of our language. It is not Turing-complete and does not allow for branching or conditionals; it simply describes the variables, equations, and relationships required by a protocol, leaving the implementation details up to the interpreter and language framework. This framework, described in the following section, provides C++ classes that parse, analyze, optimize, and interpret ZKPDL programs, employing many common compiler techniques (e.g., constant substitution and propagation, type-checking, providing error messages when undefined variables are used, etc.) in the process. We are able to understand and transform mathematical expressions into forms that provide better performance (e.g., through techniques for fixed-base exponentiation), and recognize relationships between values to be proved in zero-knowledge. All of these low-level optimizations, as well as our high-level primitives, should enable a programmer to quickly implement and evaluate the efficiency of a protocol.

We also provide a number of C++ classes that wrap ZKPDL programs into interfaces for generating and verifying proofs, as well as marshaling them between computers. We build upon these wrappers to additionally provide Cashlib, a collection of interfaces that allows a programmer to assume the role of buyer, seller, bank, or arbiter in a fair exchange system based on endorsed e-cash [27], as seen in Figure 1 and described in Section 5.3.

4 Implementation of ZKPDL

To enable implementation of the cryptographic primitives discussed in Section 2, we have designed a programming language for specifying zero-knowledge protocols, as well as an interpreter for this language. The interpreter is implemented in C++ and consists of approximately 6000 lines of code. On the prover side, the interpreter will output a zero-knowledge proof for the relations specified in the program; on the verifier side, the interpreter will be given a proof and verify whether or not it is correct. Therefore, the output of the interpreter depends on the role of the user, although the program provided to the interpreter is the same for both.

4.1 Overview

Here we provide a brief overview of some fundamental language features to give an idea of how programs are written; a full grammar for our language, containing all of its features, can be found in our documentation available online, and further sample programs can be found in Section 5. A program can be broken down into two blocks: a computation block and a proof block. Each of these blocks is optional: if a user just wants a calculator for modular (or even just integer) arithmetic then he will specify just the computation block; if, on the other hand, he has all the input values pre-computed and justs wants a zero-knowledge proof of relations between these values, he will specify just the proof block. Here is a sample program written in our language (indentations are included for readability, and are not required syntax).

sample.zkp

```
 1 computation: // compute values required for proof
 2   given: // declarations
 3     group: G = <g,h>
 4     exponents in G: x[2:3]
 5   compute: // declarations and assignments
 6     random exponents in G: r[1:3]
 7     x_1 := x_2 * x_3
 8     for(i, 1:3, c_i := g^x_i * h^r_i)
 9
10 proof:
11   given: // declarations of public values
12     group: G = <g,h>
13     elements in G: c[1:3]
14     for(i, 1:3, commitment to x_i: c_i = g^x_i * h^r_i)
15   prove knowledge of: // declarations of private values
16     exponents in G: x[1:3], r[1:3]
17   such that: // protocol specification; i.e. relations
18     x_1 = x_2 * x_3
```

In this example, we are proving that the value x_1 contained within the commitment c_1 is the product of the two values x_2 and x_3 contained in the commitments c_2 and c_3. The program can be broken down in terms of how variables are declared and used, and the computation and proof specifications. Note that some lines are repeated across the computation and proof blocks, as both are optional and hence considered independently.

4.1.1 Variable declaration

Two types of variables can be declared: group objects and numerical objects. Names of groups must start with a letter and cannot have any subscripts; sample group declarations can be seen in lines 3 and 12 of the above program. In these lines, we also declare the group generators, although this declaration is optional (as we will see later on in Section 5, it is also optional to name the group modulus).

Numerical objects can be declared in two ways. The first is in a list of variables, where their type is specified by the user. Valid types are `element`, `exponent` (which refer respectively to elements within a finite-order group and the corresponding exponents for that group), and `integer`; it should be noted that for the first two of these types a corresponding group must also be specified in the type information (see lines 4 and 13 for an example). The other way in which variables can be declared is in the compute block, where they are declared as they are being assigned (meaning they appear on the left-hand side of an equation), which we can see in lines 7 and 8. In this case, the type is inferred by the values on the right-hand side of the equation; a compile-time exception will be thrown if the types do not match up (for example, if elements from two different groups are being multiplied).

Numerical variables must start with a letter and are allowed to have subscripts.

4.1.2 Computation

The computation block breaks down into two blocks of its own: the `given` block and the `compute` block. The `given` block specifies the parameters, as well as any values that have already been computed by the user and are necessary for the computation (in the example, the group `G` can be considered a system parameter and the values `x_2` and `x_3` are just needed for the computation).

The `compute` block carries out the specified computations. There are two types of computations: picking a random value, and defining a value by setting it equal to the right-hand side of an equation. We can see an example of the former in line 6 of our sample program; in this case, we are picking three random exponents in a group (note `r[1:3]` is just syntactic sugar for writing `r_1, r_2, r_3`). We also support picking a random integer from a specified range, and picking a random prime of a specified length (examples of these can be found in Section 5). As already noted, lines 7 and 8 provide examples of lines for computing equations. In line 8, the `for` syntax is again just syntactic sugar; this time to succintly specify the relations `c_1 = g^x_1*h^r_1`, `c_2 = g^x_2*h^r_2`, and `c_3 = g^x_3*h^r_3`. We have a similar `for` syntax for specifying products or sums (much like $\prod$ or $\sum$ in conventional mathematical notation), but neither of these `for` macros should be confused with a for loop in a conventional programming language.

4.1.3 Proof specification

The proof block is comprised of three blocks: the `given` block, the `prove knowledge of` block, and the `such that` block. In the `given` block, the parameters for the proof are specified, as well as the public inputs known to both the prover and verifier for the zero-knowledge protocol. In the `prove knowledge of` block, the prover's private inputs are specified. Finally, the `such that` block specifies the desired relations between all the values; the zero-knowledge proof will be a proof that these relations are satisfied. We currently support four main types of relations:

- Proving knowledge of the opening of a commitment [67]. We can prove openings of Pedersen [65] or Fujisaki-Okamoto commitments [42, 37]. In both cases we allow for commitments to multiple values.
- Proving equality of the openings of different commitments. Given any number of commitments, we can prove the equality of any subset of the values contained within the commitments.
- Proving that a committed value is the product of two other committed values [37, 18]. As seen in our sample program, we can prove that a value x contained within a commitment is the product of two other values y, z contained within two other commitments; i.e., $x = y \cdot z$. As a special case, we can also prove that $x = y^2$.
- Proving that a committed value is contained within a public range [18, 55]. We can prove that the value x contained within a given commitment satisfies $lo \leq x < hi$, where lo and hi are both public values.

There are a number of other zero-knowledge proof types (e.g., proving a value is a Blum integer, proving that committed values satisfy some polynomial relationship, etc.), but we chose these four based on their wide usage in applications, in particular in e-cash and anonymous credentials. We note, however, that adding other proof types to the language should require little work (as mentioned in Section 4.2), as we specifically designed the language and interpreter with modularity in mind.

4.1.4 Sample usage

In addition to showing a sample program, we would also like to demonstrate a sample usage of our interpreter API. In order to use the sample ZKPDL program from Section 4.1, one could use the following C++ code (assuming there are already numerical variables named `x2` and `x3`, and a group named `G`):

```
group_map g;
variable_map v;
g["G"] = G;
v["x_2"] = x2;
v["x_3"] = x3;
InterpreterProver prover;
// compiles program with groups
prover.check("sample.zkp", g);
// computes intermediate values needed for proof
prover.compute(v);
// computes and outputs proof
ProofMessage proofMsg = (prover.getPublicVariables(),
                         prover.computeProof());
```

The method is the same for all programs: any necessary groups and/or variables are inserted into the appropriate maps, which are then passed to the interpreter. Note that the group map in this case is passed to the interpreter at "compile time" so that it may pre-compute powers of group generators to be used for exponentiation optimizations (described in the next section); however, both the group and variable maps may be provided at "compute time." Any syntactic errors will be caught at compile time, but if the inputs provided at compute time are not valid for the relations being proved, the proof will be computed anyway and the error will be caught by the verifier. The `ProofMessage` is a serializable container for the zero-knowledge proof and any intermediate values (e.g., commitments and group bases) that the verifier might need to verify the proof.

The method is almost identical for the verifier:

```
group_map g;
variable_map v;
g["G"] = G;
InterpreterVerifier verifier;
verifier.check("sample.zkp", g);
verifier.compute(v, proofMsg.publics);
bool verified = verifier.verify(proofMsg.proof);
```

As we can see, the main difference is that the verifier uses both its own public inputs and the prover's public values at compute time (with its own inputs always taking precedence over the `ProofMessage` inputs), but still takes in the proof to be checked afterwards so that the actions of the prover and verifier remain symmetric.

4.2 Optimizations

In our interpreter, we have incorporated a number of optimizations that make using our language not only more convenient but also more efficient. Here we describe the most significant optimizations, which include removing any redundancy when multiple proofs are combined and performing multi-exponentiations on cached bases when the same bases are used frequently. Other improvements specific to existing protocols can be found in Section 5.

4.2.1 Translation

To eliminate redundancy between different proofs, we first translate each proof described in Section 4.1.3 into a "fundamental discrete logarithm form." In this form, each proof can be represented by a collection of equations of the form $A = B^x \cdot C^y$. For example, if the prover would like to prove that the value x contained within $C_x = g^x h^{r_x}$ is equal to the product of the values y and z contained within $C_y = g^y h^{r_y}$ and $C_z = g^z h^{r_z}$ respectively, this is equivalent to a proof of knowledge of the discrete logarithm equalities $C_y = g^y h^{r_y}$ and $C_x = C_y^z h^{r_x - zr_y}$.

Our sample program in the previous section is first translated into this discrete logarithm form. During runtime, the values provided to the prover are then used to generate the zero-knowledge proof. In addition to eliminating redundancy between proofs of different relations in the program, this technique also allows our language to easily add new types of proofs as they become available. To add any proof that can be broken down into this discrete logarithm form, we need to add only a translation function and a rule in the grammar for how we would like to specify this proof in a program, and the rest of the work will be handled by our existing framework.

4.2.2 Multi-exponentiation

The computational performance of many cryptographic protocols, especially those used by our library, is often dominated by the need to perform many modular exponentiation operations on large integers. These operations typically involve the use of systems parameters as bases, with exponents chosen at random or provided as private inputs (e.g., Pedersen commitments, which require computation of $g^x \cdot h^r$, where g and h are publicly known). Algorithms for simultaneous multiple exponentiation allow the result of multi-base exponentiations such as these to be computed without performing each intermediate exponentiation individually; an overview of these protocols can be found in Section 14.6 of Menezes et al. [59].

Our interpreter leverages the descriptions of mathematical expressions in ZKPDL programs to recognize when fixed-base exponentiation operations occur, allowing it to precompute lookup tables at compile time that can speed up these computations dramatically. In addition to single-table multi-exponentiation techniques (i.e., the 2^w-ary method [59]), we offer programmers who expect to run the same protocol many times the ability to take advantage of time/space tradeoffs by generating large lookup tables of precomputed powers. This allows a programmer to choose parameters that balance the memory requirements of the interpreter against the need for fast exponentiation.

For single-base exponentiation, we employ window-based precomputation techniques similar to those used by PBC [57] to cache powers of fixed bases. For multi-base exponentiation of k exponents, we currently extend the 2^w-ary method to store 2^{kw}-sized lookup tables for each w-bit window of the expected exponent length, so that multi-exponentiations on exponents of length n require only n/w multiplications of stored values. While we are also evaluating other algorithms offering similar time-space tradeoffs, we demonstrate the performance gains afforded by these techniques later in Table 1.

4.2.3 Interpreter caching

We also cache the parsed, compiled environments of ZKPDL programs when they are first run. Because we accept system parameters at compile time, we are able to evaluate and propagate any subexpressions made up of fixed constants and perform exponentiation precomputations before these expressions are fully evaluated at runtime. Even without the use of large tables for fixed-based exponentiation, this optimization proves useful when repeated executions of the same program must be performed; e.g., for a bank dealing with e-coin deposits. In this case, a bank must invoke the interpreter for each coin deposited; looking ahead to Table 1 we see that, on average, this operation takes the bank 83ms. If our program were re-parsed each time, it would take an extra 10ms, as opposed to the fraction of a millisecond required to load a cached interpreter environment, saving the bank approximately 10% of computation time per transaction by avoiding parsing overhead.

5 Sample Programs and Performance

Using our language, we have written programs for a wide variety of cryptographic primitives, including blind signatures [25], verifiable encryption [28], and endorsed e-cash [27]. In the following sections, we provide our programs for these three primitives; in addition, performance benchmarks for all of them can be found at the end of the section.

5.1 CL signatures

Using our language, we have implemented the blind signature scheme due to Camenisch and Lysyanskaya [25]; as we will see in Section 5.3, CL signatures are integral to endorsed e-cash. Briefly, a *blind signature*, as introduced by Chaum [29], enables a signature issuer to sign a message without learning the contents of the message. A CL signature works in two main phases: an issuing phase, in which a user actually obtains the signature, and a proving phase, in which the user is able to prove (in zero-knowledge) to other users that he does in fact possess a valid CL signature.

The issuing phase is a one-round interaction between the recipient and the issuer, at the end of which the recipient obtains the blind signature on her message(s). Because the protocol is interactive, we present one program for each stage of the protocol. At the end of this first stage, the signature issuer will return a *partial signature* to the recipient, who will then use this signature to compute the full signature on the hidden message.

cl-recipient-proof.zkp

```
computation:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    exponents in pkGroup: x[1:L]
    integers: stat, modSize
  compute:
    random integer in [0,2^(modSize+stat)): vprime
    C := hprime^vprime * for(i, 1:L, *, gprime_i^x_i)

proof:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    group: comGroup = <f, g, h, h1, h2>
    element in pkGroup: C
    elements in comGroup: c[1:L]
       for(i, 1:L, commitment to x_i: c_i=g^x_i*h^r_i)
    integer: l_x
  prove knowledge of:
    integers: x[1:L]
    exponents in comGroup: r[1:L]
    exponent in pkGroup: vprime
  such that:
    for(i, 1:l, range: (-(2^l_x-1)) <= x_i < 2^l_x)
    C = hprime^vprime * for(i, 1:L, gprime_i^x_i)
    for(i, 1:L, c_i = g^x_i * h^r_i)
```

Next, the issuer must prove the partial signature is computed correctly, as in the following program.

cl-issuer-proof.zkp

```
computation:
  given:
    group: pkGroup = <f, g[1:L+k], h>
    element in pkGroup: C
    exponents in pkGroup: x[1:k+L]
    integers: stat, modSize, lx
  compute:
    random integer in [0,2^(modSize+lx+stat)): vpp
    random prime of length lx+2: e
    einverse := 1/e
    A := (f*C*h^vpp * for(i,L+1:k+L,*,g_i^x_i))^einverse

proof:
  given:
    group: pkGroup = <f, g[1:L+k], h>
    elements in pkGroup: A, C
    exponents in pkGroup: e, vpp, x[L+1:k]
  prove knowledge of:
    exponents in pkGroup: einverse
  such that:
    A = (f*C*h^vpp * for(i,L+1:k+L,*,g_i^x_i))^einverse
```

Once the recipient obtains the partial signature, she can unblind it to obtain a full signature; this step completes the issuing phase.

Now, the owner of a CL signature needs a way to prove that she has a signature, without revealing either the signature or the values. To accomplish this, the prover first randomizes the CL signature and then attaches a zero-knowledge proof of knowledge that the randomized signature corresponds to the original signature on the committed message.

cl-possession-proof.zkp

```
computation:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    element in pkGroup: A
    exponents in pkGroup: e, v, x[1:L]
    integers: modSize, stat
  compute:
    random integers in [0,2^(modSize+stat)): r, r_C
    vprime := v + r*e
    Aprime := A * hprime^r
    C := h^r_C * for(i, 1:L, *, gprime_i^x_i)
    D := for(i, L+1:L+k, *, gprime_i^x_i)
    fCD := f * C * D

proof:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    group: comGroup = <f, g, h, h1, h2>
    elements in pkGroup: C, D, Aprime, fCD
    elements in comGroup: c[1:L]
       for(i, 1:L, commitment to x_i: c_i=g^x_i*h^r_i)
    exponents in pkGroup: x[L+1:L+k]
    integer: l_x
  prove knowledge of:
    integers: x[1:L]
    exponents in comGroup: r[1:L]
    exponents in pkGroup: e, vprime, r_C
  such that:
    for(i, 1:L, range: (-(2^l_x - 1)) <= x_i < 2^l_x)
    C = hprime^r_C * for(i, 1:l, *, gprime_i^x_i)
    for(i, 1:L, c_i = g^x_i * h^r_i)
    fCD = (Aprime^e) * hprime^(r_C - vprime)
```

5.2 Verifiable encryption

Briefly, verifiable encryption consists of a ciphertext under the public key of some trusted third party (in our case, the arbiter) and a zero-knowledge proof that the values inside the ciphertext satisfy some relation; this pair is often referred to as a *verifiable escrow*. Our implementation of verifiable encryption is based on the construction of Camenisch and Shoup [28]. The main use of verifiable encryption in e-cash is to allow a user to verifiably encrypt the opening of a commitment under the public key of the arbiter. A recipient of such a verifiable escrow can then verify that the encrypted values correspond to the opening of the commitment.

verifiable-encryption.zkp

```
computation:
  given:
    group: secondGroup = <g[1:m], h>
    group: RSAGroup
    modulus: N
    group: G
    group: cashGroup = <f_3, gprime, hprime, f_1, f_2>
    exponents in G: x[1:m]
    elements in G: u[1:m], v, w
  compute:
    random integer in [0,N/4): s
    random exponents in secondGroup: r[1:m]
    for(i, 1:m, c_i := g_1^x_i * g_2^r_i)
    Xprime := for(i, 1:m, *, g_i^x_i) * h^s
    vsquared := v^2
    wsquared := w^2
    for(i, 1:m, usquared_i := u_i^2)

proof:
  given:
    group: secondGroup = <g[1:m], h>
    group: G
    group: RSAGroup
    modulus: N
    group: cashGroup = <f_3, gprime, hprime, f_1, f_2>
    element in cashGroup: X
    elements in secondGroup: Xprime, c[1:m]
    for(i,1:m,commitment to x_i: c_i=g_1^x_i*g_2^r_i)
    elements in G: a[1:m], b, d, e, f, usquared[1:m],
                   vsquared, wsquared
  prove knowledge of:
    integers: x[1;M], r
    exponent in G: hash
    exponents in secondGroup: r[1:m], s
  such that:
    for(i, 1:m, range: -N/2 + 1 <= x_i < N/2)
    vsquared = f^(2*r)
    wsquared = (d * e^hash)^(2*r)
    for(i, 1:m, usquared_i = b^(2*x_i) * a_i^(2*r))
    X = for(i, 1:m, *, f_i^x_i)
    Xprime = for(i, 1:m, *, g_i^x_i) * h^s
```

5.3 E-cash

Electronic cash, or e-cash for short, was first introduced by Chaum [29] and can be thought of as the electronic equivalent of cash; i.e., an electronic currency that preserves users' anonymity, as opposed to electronic checks [31] or credit cards. We implement endorsed e-cash, due to Camenisch, Lysyanskaya, and Meyerovich [27] (which is an extension of compact e-cash [22]), for two main reasons. Our first reason is that an endorsed e-coin can be split up into two parts, its endorsement and an unendorsed component; only with both of these parts can the coin be considered complete. As we will see in Section 6.2.1, this property enables efficient fair exchange. The second reason for choosing endorsed e-cash is that it is *offline*, which means the bank does not need to be active in every transaction; this significantly reduces the burden placed on the bank. Although the bank does not check the coin in every interaction, endorsed e-cash has the property that double-spenders (i.e., users who try to spend the same coin twice) can be caught by the bank at the time of deposit and punished accordingly. Because e-cash is meant to preserve privacy, however, a user is also guaranteed that unless she double spends a coin, her identity will be kept secret.

During the withdrawal phase of endorsed e-cash, a user contacts the bank. Before withdrawing, the user will have registered with the bank by storing a commitment. In order to prove her identity, then, the user will provide a proof that she knows the opening of the registered commitment. This can be accomplished using the following simple program:

user-id-proof.zkp

```
proof:
  given:
    group: cashGroup = <f,g,h,h1,h2>
    elements in cashGroup: A, pk_u
      commitment to sk_u: A = g^sk_u * h^r_u
  prove knowledge of:
    exponents in cashGroup: sk_u, r_u
  such that:
    pk_u = g^sk_u
    A = g^sk_u * h^r_u
```

Once the bank has verified this proof, the user and the bank will run a protocol to obtain a CL signature (using the programs we saw in Section 5.1) on the user's identity and two pseudo-random function seeds. These private values and the signature on them define a wallet that contains W coins (where W is a system-wide public parameter).

When a user wishes to spend one of her coins, she splits it up into its unendorsed part and the endorsement. She then sends the unendorsed component to a merchant and proves it is valid. If the merchant then sends her what she wanted to buy, she will follow up with the endorsement to complete the coin and the transaction is complete. The following program is used for proving the validity of a coin.

coin-proof.zkp

```
computation:
  given:
    group: cashGroup = <f, g, h, h1, h2>
    exponents in cashGroup: s, t, sk_u
    integer: J
  compute:
    random exponents in cashGroup: r_B, r_C, r_D, x1,
                                   x2, r_y, R
    alpha := 1 / (s + J)
    beta := 1 / (t + J)
    C := g^s * h^r_C
    D := g^t * h^r_D
    y := h1^x1 * h2^x2 * f^r_y
    B := g^sk_u * h^r_B
    S := g^alpha * g^x1
    T := g^sk_u * (g^R)^beta * g^x2

proof:
  given:
    group: cashGroup = <f, g, h, h1, h2>
    elements in cashGroup: y, S, T, B, C, D
       commitment to sk_u: B = g^sk_u * h^r_B
       commitment to s: C = g^s * h^r_C
       commitment to t: D = g^t * h^r_D
    integer: J
  prove knowledge of:
    exponents in cashGroup: x1, x2, r_y, sk_u, alpha,
          beta, s, t, r_B, r_C, r_D, R
  such that:
    y = h1^x1 * h2^x2 * f^r_y
    S = g^alpha * g^x1
    T = g^sk_u * (g^R)^beta * g^x2
    g = (g^J * C)^alpha * h^(-r_C / (s+J))
    g = (g^J * D)^beta * h^(-r_D / (t+J))
```

5.4 Performance

Here we measure the communication and computational resources used by our system when running each of the programs above. The benchmarks presented in Table 1 were collected on a MacBook Pro with a 2.53GHz Intel Core 2 Duo processor and 4GB of RAM running OS X 10.6; we therefore expect that these results will reflect those of a typical home user with no special cryptographic hardware support.

As for speed, caching exponents of fixed bases results in a significant performance increase, making it an important optimization for applications that require repeated protocol executions. The only caveat is that the exponent cache required for complex protocols can grow to hundreds of megabytes (using faster-performing parameters), and so our library allows users to choose whether to use caching, and if so how much of the cache should be used by this optimization.

The time taken for the higher-level protocols provides a clear view of the complexity of each protocol. For example, the marked difference between the time required to generate a CL issuer proof and a CL possession proof can be attributed to the fact that a CL issuer proof requires proving only one discrete log relation, while a CL possession proof on three private values requires three range proofs and five more discrete log relations.

Table 1 also shows that verifiable encryption is by far the biggest bottleneck, requiring almost three times as much time to compute as any other step. As seen in the program in Section 5.2, there is one range proof performed for each value contained in the verifiable escrow. In order to perform a range proof, the value contained in the range must be decomposed as a sum of four squares [66]. Because the values used in our verifiable encryption program are much larger than the ones used in CL signatures (about 1024 vs. 160 bits, to get 80-bit security for both), this decomposition often takes considerably more time for verifiable encryption than it does for CL signatures. Furthermore, since the values being verifiably encrypted are different each time, caching the decomposition of the values wouldn't be of any use.

A final observation on computational performance is that proving possession of a CL signature completely dominates the time required to prove the validity of a coin, since the timings for the two proofs are within milliseconds. This suggests that the only way to significantly improve the performance of e-coins and verifiable encryption would be to develop more efficient techniques for range proofs (which has in fact been the subject of some recent cryptographic research [49, 19, 68]).

In terms of proof size, range proofs are much larger than proofs for discrete logarithms or multiplication. This is to be expected, as translating a range proof into discrete logarithm form (as described in Section 4.2) requires eleven equations, whereas a single DLR proof requires only one, and a multiplication proof requires two.

6 Implementation of Cashlib

Using the primitives described in the previous section, we wrote a cryptographic library designed for optimistic fair exchange protocols. Fair exchange [32] involves a situation in which a buyer wants to make sure that she doesn't pay a merchant unless she gets what she is buying, while the merchant doesn't want to give away his goods unless he is guaranteed to be paid. It is known that fair exchange cannot be done without a trusted third party [63], but *optimistic* fair exchange [3, 4] describes the cases in which the trusted third party has to get involved only in the case of a dispute.

The library was written in C++ and consists of approximately 11000 lines of code in addition to the interpreter. A previous version of the library in which all the protocols and proofs were hand-coded (i.e., the interpreter was not used) consisted of approximately 20000 lines of code, meaning that the use of roughly 400 lines of ZKPDL was able to replace 9000 lines of our original C++ code (and, as we will see, make our operations more efficient as well).

6.1 Endorsed e-cash

A description of endorsed e-cash can be found in Section 5.3; the version used in our library, however, contains a number of optimizations. Just as with real cash, we now allow for different coin denominations. Each coin denomination corresponds to a different bank public key, so once the user requests a certain denomination, the wallet is then signed using the corresponding public key. A coin generated from such a wallet will verify only when the same public key of the bank is used, and thus the merchant can check for himself the denomination of

Program type	Prover (ms)		Verifier (ms)		Proof size	Cache size	Multi-exps	
	With cache	Without	With cache	Without	(bytes)	(Mbytes)	Prover	Verifier
DLR proof	3.07	3.08	1.26	1.25	511	0	2	1
Multiplication proof	2.03	4.07	1.66	2.32	848	33.5	8	2
Range proof	36.36	74.52	21.63	31.54	5455	33.5	31	11
CL recipient proof	119.92	248.31	70.76	112.13	19189	134.2	104	39
CL issuer proof	7.29	7.38	1.73	1.73	1097	0	2	1
CL possession proof	125.89	253.17	78.19	117.67	19979	134.2	109	40
Verifiable encryption	416.09	617.61	121.87	162.77	24501	190.2	113	42
Coin	134.37	271.34	83.01	121.83	22526	223.7	122	45

Table 1: Time (in milliseconds) and size (in bytes) required for each of our proofs, averaged over twenty runs. Timings are considered from both the prover and verifier sides, as are the number of multi-exponentiations, and are considered both with and without caching for fixed-based exponentiations; the size of the cache is also measured (in megabytes). As we can see, using caching results on average in a 48% speed improvement for the prover, and a 31% improvement for the verifier.

the coin.

The program in Section 5.3 also reflects our decision to randomize the user's spending order rather than having them perform a range proof that the coin index was contained within the proper range. As the random spending order does not reveal how many coins are left in the wallet, the user's privacy is still protected even though the index is publicly available. Furthermore, because range proofs are slow and require a fair amount of space (see Table 1 for a reminder), this optimization resulted in coins that were 20% smaller and 21% faster to generate and verify.

Finally, endorsed e-cash requires a random value contributed by both the merchant and the user. Since e-coin transactions should be done over a secure channel, in practice we expect that SSL connections will be used between the user and the merchant. One useful feature of an SSL connection is that it already provides both parties with shared randomness, and thus this randomness can be used in our library to eliminate the need for a redundant message.

6.2 Buying and Bartering

Our library implements two efficient optimistic fair exchange protocols for use with e-cash. Belenkiy et al. [10] provide a *buy* protocol for exchanging a coin with a file, while Küpçü and Lysyanksaya [52] provide a *barter* protocol for exchanging two files or blocks. The two protocols serve different purposes (buy vs. barter) and so we have implemented both.

Two of the main usage scenarios of fair exchange protocols are e-commerce and peer-to-peer file sharing [10]. In e-commerce, one needs to employ a buy protocol to ensure that both the user and the merchant are protected; the user receives her item while the merchant receives his payment. In a peer-to-peer file sharing scenario, peers exchange files or blocks of files. In this setting, it is more beneficial to barter for the blocks than to buy them one at a time; for an exchange of n blocks, buying all the blocks requires $O(n)$ verifiable escrow operations (which, as discussed in Section 5.4, are quite costly), whereas bartering for the blocks requires only one such operation, regardless of the number of blocks exchanged.

Although the solution might seem to be to barter all the time and never buy, Belenkiy et al. suggest that both protocols are useful in a peer-to-peer file sharing scenario. Peers who have nothing to offer but would still like to download can offer to buy the files, while peers who would like only to upload and have no interest in downloading can act as the merchant and earn e-cash. Due to the resource considerations mentioned above, however, bartering should always be used if possible.

Because peers do not always know beforehand if they want to buy or barter for a file, we have modified the buy protocol to match up with the barter protocol in the first two messages. This modification, as well as outlines of both the protocols, can be seen in Figure 3. We further modified both protocols to let them exchange multiple blocks at once, so that one block of the fair exchange protocol might correspond to multiple blocks of the underlying file.

We give an overview of each protocol below, with the optimizations we have added. We have also implemented the trusted third parties (the bank and the arbiter) necessary for e-cash and fair exchange. Although we do not describe in detail the resolution and bank interaction protocols, these can be found in the original papers [10, 52] and we provide performance benchmarks for the bank in Table 2.

6.2.1 Buying

The modified buy protocol is depicted on the left in Figure 3, although we also allow for the users to participate in the original buy protocol (in which the messages appear in a slightly different order). To initiate the modified buy protocol, the buyer sends a "setup" message, which consists of an unendorsed coin and a verifiable escrow on its corresponding endorsement. Upon receiving

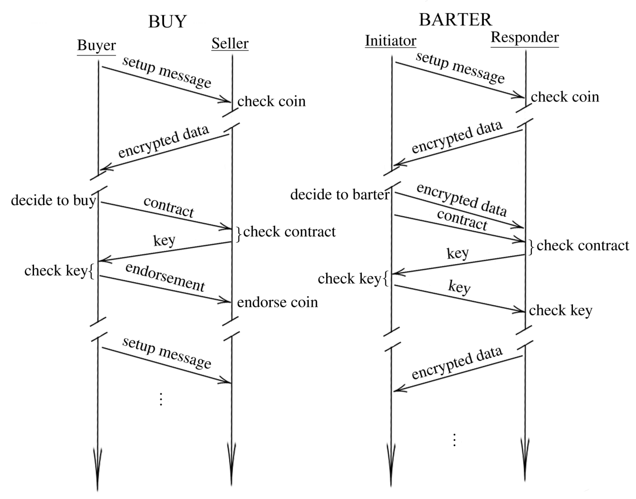

Figure 3: This figure provides outlines of both our buy and barter protocols [10, 52]. Until the decision to buy or barter, the two protocols are identical; the main difference is that in a buy protocol, the setup message must be sent for each file exchange, which results in a linear efficiency loss as compared to bartering.

this message, the seller will use the programs in Section 5 to check the validity of the coin and the escrow. If these proofs verify, the seller will proceed by sending back an encrypted version of his file (or file block). Upon receiving this ciphertext, the buyer will store it (and a Merkle hash of it, for use with the arbiter in case the protocol goes wrong later on) and send back a contract, which consists of a hash of the seller's file and some session information. The seller will check this contract and, if satisfied with the details of the agreement, send back its decryption key. The buyer can then use this key to decrypt the ciphertext it received in the second message of the protocol. If the decryption is successful, the buyer will send back his endorsement on the coin. If in these last steps either party is unsatisfied (for example, the file does not decrypt or the endorsement isn't valid for the coin from the setup message), they can proceed to contact the arbiter and run resolution protocols [10].

6.2.2 Bartering

This protocol is depicted on the right in Figure 3; because the first two messages of the barter protocol (the setup message and the encrypted data) are identical to those in the buy protocol described in the previous section, we do not describe them again here and instead jump directly to the third message. Because bartering involves an exchange of data, the initiator will respond to the receipt of the ciphertext with a ciphertext of her own, corresponding to an encryption of her file. She will also send a contract, which is similar to the buy contract but also contains hash information for her file. The responder will then check this contract as the seller did in the buy protocol, and if satisfied with the agreement will send back his decryption key. If the ciphertext decrypts correctly (i.e., decrypts to the file described in the contract) then the initiator can respond in turn with her own decryption key. If this decryption key is also valid, both parties have successfully obtained the desired files and the barter protocol can be considered complete. If neither party had to contact the arbiter (for similar reasons as in the buy protocol; i.e., a file did not decrypt correctly) then they are free to engage in future barter protocols without the overhead of an additional setup message. Otherwise, they need to resolve with the arbiter [52].

6.3 Library performance

In Table 2, we can see the computation time and size complexity for the steps described above, as well as computation and communication overhead for the withdraw and deposit protocols involving the bank. The numbers in the table were computed on the same computer as those in Section 5.4.

The numbers in Table 2 clearly demonstrate our earlier observation that bartering is considerably more efficient than buying, both in terms of computation and communication overhead. The setup message for both buying and bartering takes about 600ms to generate and approximately 46kB of space. In contrast, the rest of the barter protocol takes very little time; on the order of milliseconds for both parties (and about 1.5kB of total overhead).

In addition, we consider the same protocols run using a previous "naïve" version of our library, which provided the same e-cash API and employed some multi-exponentation optimizations, but did not use ZKPDL. Using the optimizations available to the interpreter is considerably faster over our previous approach, meaning that our interpreter has not only made developing our protocols more convenient, but has also helped to improve efficiency.

7 Related work

Similar to our approach, FairPlayMP [14] (and its predecessor, FairPlay [58]) provides a language-based system for secure multi-party computation, allowing multiple parties to jointly compute a function on private inputs while revealing nothing but the resulting value. At the heart of FairPlayMP is a programming language, SFDL 2.0 (short for Secure Function Definition Language), that allows programmers to specify a multi-party computation. The authors provide a compiler that transforms SFDL programs into boolean circuits, and an engine that securely evaluates these circuits and distributes the resulting values among the involved parties. Although this

Operation	Time (ms)	"Naïve" time (ms)	Size (B)
Withdraw (user)	126.35	290.79	20093
Withdraw (bank)	83.36	140.02	1167
Deposit (bank)	82.11	128.36	22526
Buying a block (buyer)	628.49	901.04	47286
Buying a block (seller)	211.89	275.94	203
Barter setup message	608.29	881.32	46934
Checking setup message	210.61	276.98	n/a
Barter after setup (initiator)	18.02	18.28	1280
Barter after setup (responder)	1.11	1.18	204

Table 2: Average time required and network overhead, in milliseconds and bytes respectively, for each stage in our e-cash implementation. The timings were averaged over twenty runs, and caching and compression optimizations were used. For the naïve timings, an older version of the library was used, which uses some multi-exponentation optimization techniques but not the interpreter; we can see a clear improvement when using ZKPDL. Parameters were used to provide a security level of 80 bits (160-bit SHA-1 hashing, 128-bit AES encryption, 1024-bit RSA moduli, and 1024-bit DSA signatures).

is a very useful tool, it uses generic circuit techniques, and thus from an efficiency standpoint it is often more desirable to develop a multi-party computation scheme specific to the intended application.

IBM's Idemix project [20, 15] has independently developed a library for zero-knowledge proofs and anonymous credentials using Java; their library provides a system for obtaining, proving, and verifying anonymous credentials for use in a privacy-preserving identity systems. While Idemix and our work both provide implementations of anonymous credentials and CL signatures, our focus on efficient, repeated executions of e-cash transactions has led us to pursue our language-based strategy and develop a performance-optimized interpreter, unlike the Idemix implementation. The CACE project, independent of our efforts, has also designed a high-level language for zero-knowledge protocols; their work has focused on a compiler that can output implementation and LaTeX code from these descriptions [6, 5], and automatically check the soundness of compiled protocols using theorem proving techniques [1].

There are also compilers available [16, 7] for the generation of proofs of security and correctness for cryptographic protocols. While this is an interesting and important area of research, these tools largely focus on static analysis of protocols rather than performance. Perhaps more similar to our work, the languages Cryptol [54] and Stupid [53] provide a simple interface for developing low-level implementations of cryptographic primitives (such as hash functions) which can then be analyzed and translated into native code on different platforms.

8 Conclusions and Future Work

In this paper we have introduced a language for generating (and verifying) widely-used zero-knowledge proofs of knowledge. Through sample programs, we have demonstrated how our language is used to express advanced cryptographic primitives such as blind signatures, verifiable encryption, and endorsed e-cash. We presented optimizations provided by our language's interpreter and have shown they provide significant benefit.

Atop our language framework, we built a library that provides optimistic fair exchange protocols based on electronic cash. We have further presented optimizations for the protocols provided by Cashlib and argued for their practicality in network-based applications.

Much future work is possible for the ZKPDL language and interpreter. There are many other cryptographic primitives which could be incorporated into the language (e.g., encryption, signatures, hash functions), and other zero-knowledge protocols that could be added as relations (e.g., alternate and "fuzzy" schemes for range proofs). Incorporating these primitives, perhaps by allowing for subroutines and the composability of ZKPDL programs, would allow our library to be more easily extended and potentially have applicability to a broader range of secure systems. The analysis of ZKPDL programs—e.g., to automatically verify protocols and identify security errors through type analysis or formal verification techniques—provides another interesting area of study.

For increased performance on multicore architectures, we are working on analyzing dependencies among the expressions evaluated by our interpreter. The simplicity of our language, e.g., in `compute` blocks, allows a coarse-grained approach, as the only dependencies that arise between lines of ZKPDL are from variables which have been declared and assigned in previous lines.

Finally, in terms of extending Cashlib, to improve a bank's efficiency it might also be possible to speed up coin verification time by supporting batch verification techniques [23, 40] for CL signatures; we leave this as one of many interesting open problems.

Acknowledgments

We gratefully acknowledge George Danezis, our shepherd, and our anonymous reviewers for their valuable feedback on earlier versions of this paper. We also would like to thank Gabriel Bender and Alex Hutter for their work developing earlier versions of Cashlib, as well as Carleton Coffrin for his assistance with ANTLR. This work is supported in part by NSF CyberTrust grant 0627553.

References

[1] Almeida, J. B., Bangerter, E., Barbosa, M., Krenn, S., Sadeghi, A.-R., and Schneider, T. A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols. In *ESORICS '10* (2010).

[2] Androulaki, E., Raykova, M., Srivatsan, S., Stavrou, A., and Bellovin, S. PAR: payment for anonymous routing. In *Privacy Enhancing Technologies Symposium (PETS)* (2008), vol. 5134 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 219–236.

[3] Asokan, N., Shoup, V., and Waidner, M. Optimistic fair exchange of digital signatures. In *Proc. Eurocrypt '98* (1998), vol. 1403 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 591–606.

[4] Avoine, G., and Vaudenay, S. Optimistic fair exchange based on publicly verifiable secret sharing. In *ACISP* (2004), vol. 3108 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 74–85.

[5] Bangerter, E., Barzan, S., Krenn, S., Sadeghi, A.-R., Schneider, T., and Tsay, J.-K. Bringing zero-knowledge proofs of knowledge to practice. In *17th International Workshop on Security Protocols* (2009).

[6] Bangerter, E., Camenisch, J., Krenn, S., Sadeghi, A.-R., and Schneider, T. Automatic generation of sound zero-knowledge protocols. Cryptology ePrint Archive, Report 2008/471, 2008. `http://eprint.iacr.org/2008/471`.

[7] Barbosa, M., Noad, R., Page, D., and Smart, N. First steps toward a cryptography-aware language and compiler. Cryptology ePrint Archive, Report 2005/160, 2005. `http://eprint.iacr.org/2005/160`.

[8] Belenkiy, M., Camenisch, J., Chase, M., Kohlweiss, M., Lysyanskaya, A., and Shacham, H. Delegatable anonymous credentials. In *Proc. Crypto '09* (2009), vol. 5677 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 108–125.

[9] Belenkiy, M., Chase, M., Erway, C., Jannotti, J., Küpçü, A., and Lysyanskaya, A. Incentivizing outsourced computation. In *NetEcon* (2008), pp. 85–90.

[10] Belenkiy, M., Chase, M., Erway, C., Jannotti, J., Küpçü, A., Lysyanskaya, A., and Rachlin, E. Making P2P accountable without losing privacy. In *WPES* (2007), ACM, pp. 31–40.

[11] Belenkiy, M., Chase, M., Kohlweiss, M., and Lysyanskaya, A. Non-interactive anonymous credentials. In *Proc. 5th Theory of Cryptography Conference (TCC)* (2008), pp. 356–374.

[12] Bellare, M., and Goldreich, O. On defining proofs of knowledge. In *Proc. Crypto '92* (1992), vol. 740 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 390–420.

[13] Bellare, M., and Rogaway, P. Random oracles are practical: a paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security (CCS) '93* (1993), pp. 62–73.

[14] Ben-David, A., Nisan, N., and Pinkas, B. FairplayMP: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security (CCS) '08* (2008), pp. 257–266.

[15] Bichsel, P., Binding, C., Camenisch, J., Gross, T., Heydt-Benjamin, T., Sommer, D., and Zaverucha, G. Cryptographic protocols of the identity mixer library, v. 1.0. IBM Research Report RZ3730, 2009.

[16] Blanchet, B., and Pointcheval, D. Automated security proofs with sequences of games. In *Proc. Crypto '06* (2006), vol. 4117 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 537–554.

[17] Blum, M., de Santis, A., Micali, S., and Persiano, G. Non-interactive zero-knowledge. *SIAM Journal of Computing 20*, 6 (1991), 1084–1118.

[18] Boudot, F. Efficient proofs that a committed number lies in an interval. In *Proc. Eurocrypt '00* (2000), vol. 1807 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 431–444.

[19] Camenisch, J., Chaabouni, R., and abhi shelat. Efficient protocols for set membership and range proofs. In *Proc. Asiacrypt '08* (2008), pp. 234–252.

[20] Camenisch, J., and Herreweghen, E. V. Design and implementation of the idemix anonymous credential system. In *ACM Conference on Computer and Communications Security (CCS) '02* (2002), ACM, pp. 21–30.

[21] Camenisch, J., Hohenberger, S., Kohlweiss, M., Lysyanskaya, A., and Meyerovich, M. How to win the clonewars: efficient periodic n-times anonymous authentication. In *ACM Conference on Computer and Communications Security (CCS) '06* (2006), pp. 201–210.

[22] Camenisch, J., Hohenberger, S., and Lysyanskaya, A. Compact e-cash. In *Proc. Eurocrypt '05* (2005), vol. 3494 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 302–321.

[23] Camenisch, J., Hohenberger, S., and Pedersen, M. Ø. Batch verification of short signatures. In *Proc. Eurocrypt '07* (2007), vol. 4515 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 246–263.

[24] Camenisch, J., and Lysyanskaya, A. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Proc. Eurocrypt '01* (2001), vol. 2045 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 93–118.

[25] Camenisch, J., and Lysyanskaya, A. A signature scheme with efficient protocols. In *Proc. SCN '02* (2002), vol. 2576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 268–289.

[26] Camenisch, J., and Lysyanskaya, A. Signature schemes and anonymous credentials from bilinear maps. In *Proc. Crypto '04* (2004), vol. 3152 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 56–72.

[27] Camenisch, J., Lysyanskaya, A., and Meyerovich, M. Endorsed e-cash. In *IEEE Symposium on Security and Privacy* (2007), pp. 101–115.

[28] Camenisch, J., and Shoup, V. Practical verifiable encryption and decryption of discrete logarithms. In *Proc. Crypto '03* (2003), vol. 2729 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 126–144.

[29] Chaum, D. Blind signatures for untraceable payments. In *Proc. Crypto '82* (1982), Lecture Notes in Computer Science, Springer-Verlag, pp. 199–203.

[30] CHAUM, D. Security without identification: transaction systems to make big brother obsolete. *Communications of the ACM 28*, 10 (1985), 1030–1044.

[31] CHAUM, D., DEN BOER, B., VAN HEYST, E., MJØLSNES, S. F., AND STEENBEEK, A. Efficient offline electronic checks (extended abstract). In *Proc. Eurocrypt '89* (1989), pp. 294–301.

[32] COX, B., TYGAR, J., AND SIRBU, M. Netbill security and transaction protocol. In *Proc. 1st Usenix Workshop on Electronic Commerce* (1995), pp. 77–88.

[33] DAEMEN, J., AND RIJMEN, V. *Rijndael: AES – The Advanced Encryption Standard*. Springer-Verlag, 2002.

[34] DAMGÅRD, I. On sigma protocols. `http://www.daimi.au.dk/ivan/Sigma.pdf`.

[35] DAMGÅRD, I. Payment systems and credential mechanism with provable security against abuse by individuals. In *Proc. Crypto '88* (1988), vol. 403 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 328–335.

[36] DAMGÅRD, I., DUPONT, K., AND PEDERSEN, M. Ø. Unclonable group identification. In *Proc. Eurocrypt '06* (2006), vol. 4004 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 555–572.

[37] DAMGÅRD, I., AND FUJISAKI, E. A statistically-hiding integer commitment scheme based on groups with hidden order. In *Proc. Asiacrypt '02* (2002), vol. 2501 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 125–142.

[38] DAMGÅRD, I., GROTH, J., AND SALOMONSEN, G. The theory and implementation of an electronic voting system. In *Proc. Secure Electronic Voting (SEC)* (2003), pp. 77–100.

[39] FEIGE, U., LAPIDOT, D., AND SHAMIR, A. Multiple non-interactive zero-knowledge proofs based on a single random string. In *Proc. 31st Symposium on Theory of Computing (STOC)* (1990), pp. 308–317.

[40] FERRARA, A. L., GREEN, M., HOHENBERGER, S., AND PEDERSEN, M. Ø. Practical short signature batch verification. In *Proc. CT-RSA* (2009), pp. 309–324.

[41] FIAT, A., AND SHAMIR, A. How to prove yourself: practical solutions to identification and signature problems. In *Proc. Crypto '86* (1986), vol. 263 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 186–194.

[42] FUJISAKI, E., AND OKAMOTO, T. Statistical zero knowledge protocols to prove modular polynomial relations. In *Proc. Crypto '97* (1997), vol. 1294 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 16–30.

[43] GENNARO, R., GENTRY, C., AND PARNO, B. Non-interactive verifiable computing: outsourcing computation to untrusted workers. Cryptology ePrint Archive, Report 2009/547, 2009. `http://eprint.iacr.org/2009/547`.

[44] GMP. The GNU MP Bignum library. `http://gmplib.org`.

[45] GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. How to construct random functions (extended abstract). In *Proc. 25th Symposium on the Foundations of Computer Science (FOCS)* (1984), pp. 464–479.

[46] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *J. ACM 38*, 3 (1991), 691–729.

[47] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof systems. In *Proc. 17th Symposium on the Theory of Computing (STOC)* (1985), pp. 186–208.

[48] GOLDWASSER, S., MICALI, S., AND RIVEST, R. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing 17*, 2 (1988), 281–308.

[49] GROTH, J. Non-interactive zero-knowledge arguments for voting. In *ACNS* (2005), vol. 3531 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 467–482.

[50] HEYDT-BENJAMIN, T., CHAE, H.-J., DEFEND, B., AND FU, K. Privacy for public transportation. In *Privacy Enhancing Technologies Symposium (PETS)* (2006), pp. 1–19.

[51] ISHIDA, N., MATSUO, S., AND OGATA, W. Divisible voting scheme. In *Proc. ISC '03* (2003), vol. 2851 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 137–150.

[52] KÜPÇÜ, A., AND LYSYANSKAYA, A. Usable optimistic fair exchange. In *Proc. CT-RSA '10* (2010), vol. 5985 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 252–267.

[53] LAURIE, B., AND CLIFFORD, B. Stupid: a meta-language for cryptography, 2010. `http://code.google.com/p/stupid-crypto`.

[54] LEWIS, J., AND MARTIN, B. Cryptol: high assurance, retargetable crypto development, and validation. In *Proc. Military Communications Conference '03* (2003), pp. 820–825.

[55] LIPMAA, H. On Diophantine complexity and statistical zero-knowledge arguments. In *Proc. Asiacrypt '03* (2003), vol. 2894 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 398–415.

[56] LIPMAA, H., ASOKAN, N., AND NIEMI, V. Secure vickrey auctions without threshold trust. In *Proc. Financial Cryptography '02* (2002), vol. 2357 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 87–101.

[57] LYNN, B. PBC (pairing-based cryptography) library. `http://crypto.stanford.edu/pbc`.

[58] MALKHI, D., NISAN, N., PINKAS, B., AND SELLA, Y. Fairplay - a secure two-party computation system. In *USENIX Security Symposium* (2004), pp. 287–302.

[59] MENEZES, A. J., VAN OORSCHOT, P., AND VANSTONE, S. *Handbook of Applied Cryptography*. CRC Press, 1997.

[60] MERKLE, R. A digital signature based on a conventional encryption function. In *Proc. Crypto '88* (1987), vol. 293 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 369–378.

[61] NAOR, M., AND YUNG, M. Universal one-way hash functions and their cryptographic applications. In *Proc. 21st Symposium on Theory of Computing (STOC)* (1989), pp. 33–43.

[62] NGUYEN, L., AND SAFAVI-NAINI, R. Dynamic k-times anonymous authentication. In *ACNS* (2005), vol. 3531 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 318–333.

[63] PAGNIA, H., AND GÄRTNER, F. On the impossibility of fair exchange without a trusted third party. Darmstadt University Technical Report TUD-BS-1999-02, 1999.

[64] PARR, T. ANTLR parser generator. `http://www.antlr.org`.

[65] PEDERSEN, T. P. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proc. Crypto '91* (1992), vol. 576 of *Lecture Notes in Computer Science*, Springer-Verlag.

[66] RABIN, M., AND SHALLIT, J. Randomized algorithms in number theory. *Communications on Pure and Applied Mathematics 39*, 1 (1986), 239–256.

[67] SCHNORR, C.-P. Efficient signature generation by smart cards. *Journal of Cryptology 4*, 3 (1991), 161–174.

[68] SCHOENMAKERS, B. Interval proofs revisited. In *International Workshop on Frontiers in Electronic Elections* (2005).

A Translations of Proofs

In Section 4.2 we described the translation of a multiplication proof into a collection of discrete logarithm equality proofs. Here, we describe the translations done for non-negative proofs, as well as the low-high range proofs that build on top of non-negative proofs.

For non-negative proofs, we would like to prove that a given committed value x is non-negative, so that $x \geq 0$. To accomplish this, we first need a commitment C_x to x. We now use a theorem of Lagrange that states that any non-negative integer can be written as a sum of (at most) four squares. To show that x is non-negative, therefore, it suffices to find and prove its decomposition into a sum of four squares. The interpreter first finds this decomposition (using a randomized algorithm due to Rabin and Shallit [66]); namely the decomposition $x = x_1^2 + x_2^2 + x_3^2 + x_4^2$. For each x_i, the interpreter now computes a commitment $C_i = g^{x_i} h^{r_i}$ to the value x_i and a commitment $C_{2i} = g^{x_i^2} h^{r_{2i}}$ to the value x_i^2, with the special property that the commitment to x_4^2 uses randomness $r_x - (r_{21} + r_{22} + r_{23})$. It now needs to perform a square proof (remember that this is just a special case of a multiplication proof) for each i. Finally, it needs to prove that this decomposition corresponds to the original x. Because of the way the randomness for C_{24} is formed, if the commitments were formed correctly then the following equality will hold: $C_x = \prod_{i=1}^{4} C_{2i}$. Note that a separate proof does not need to be formed for this equality, since the verifier can check it directly given the commitments C_x and C_{2i} (due to the homomorphic property of the commitment scheme). This means that the final collection of discrete logarithm equalities needed to prove that $x \geq 0$ will be four square proofs corresponding to the values x_i and x_i^2 contained in C_i and C_{2i} respectively (for the explicit form of the discrete log equation for square proofs, refer back to Section 4.2.

Once we have non-negative proofs, low-high range proofs follow almost directly. First, observe that to prove $lo \leq x < hi$ for some value x contained in a commitment C_x, it suffices to show that $x - lo \geq 0$ and $x - hi \geq 0$. In fact, we can make this proof even more efficient by further observing that we can just prove $(x - lo)(hi - x) \geq 0$, thus eliminating one non-negative proof (note that we can do this because it is not possible for both of these quantities to be negative at the same time). Therefore, we first need to form commitments to $x - lo$ and $hi - x$. Because the verifier knows the commitment C_x as well as the bounds lo and hi, he can compute the commitments C_{x-lo} and C_{hi-x} directly, as $C_{x-lo} = C_x \cdot g^{-lo}$ and $C_{hi-x} = g^{hi} \cdot C_x^{-1}$. The prover will then need to form a commitment C_{prod} to the product $(x - lo)(hi - x)$ and, using a multiplication proof, prove that the value contained within C_{prod} is the product of the values contained in C_{x-lo} and C_{hi-x}. Finally, he will need to prove (using the translations we just saw above) that the value in C_{prod} is non-negative.

B ZKPDL Grammar

To enhance the brief overview of our language given in Section 4, we provide here a full EBNF grammar for our language.

```
spec := (computation)?  (proof)?
computation := “computation” COLON “given” COLON givenList “compute” COLON computeRandomList
computeEquationList
proof := “proof” COLON “given” COLON givenList “prove” “knowledge” “of” COLON knowledgeList “such”
“that” COLON suchThatList
suchThatList := (suchThatRel)+
suchThatRel := equalRelation | rangeRelation | forRelation
knowledgeList := (exponentDecl | integerDecl)+
givenList := (groupDecl | elementsDecl | exponentDecl | integerDecl)+
randomBndDecl := (“integer” | “integers”) “in” LBRACKET expr COMMA expr RPAREN COLON idGeneralDeclList
randomPrimeDecl := (“prime” | “primes”) “of” “length” expr COLON idGeneralDeclList
computeRandomList := (“random” (randExponentDecl | randomPrimeDecl | randomBndDecl))*
groupDecl := “group” COLON identifierDecl (EQUAL setDecl)?  (“modulus” COLON subscriptIdentifierDecl)?
randExponentDecl := (“exponent” | “exponents”) “in” identifier COLON idGeneralDeclList
exponentDecl := (“exponent” | “exponents”) “in” identifier COLON idGeneralDeclList
elementsDecl := (“element” | “elements”) “in” identifier COLON idGeneralDeclList (elementsEquationList)?
integerDecl := (“integer” | “integers”) COLON idGeneralDeclList
computeEquationList := (computeEquation)+
computeEquation := equalDeclRelation | comRelation | forRelation
forRelation := “for” LPAREN identifier COMMA expr COLON expr COMMA (rangeRelation | genEqual) RPAREN
forCom := “for” LPAREN identifier COMMA expr COLON expr COMMA comRelation RPAREN
elementsEquationList := (elementsEquation)+
elementsEquation := comRelation | forCom
comRelation := “commitment” “to” idSubList COLON subscriptIdentifier EQUAL expr
genEqual := identifier (SUBSCRIPT (ID | INTLIT))? (EQUAL | CEQUAL) expr
equalRelation := subscriptIdentifier EQUAL expr
equalDeclRelation := subscriptIdentifierDecl CEQUAL expr
rangeRelation := “range” COLON expr (LTHAN | LEQ) expr (LTHAN | LEQ) expr | (GTHAN | GEQ) expr (GTHAN
| GEQ) expr
expr := prodExpr (ADD prodExpr | SUB prodExpr)*
forExpr := “for” LPAREN identifier COMMA expr COLON expr COMMA (ADD | MUL) COMMA expr RPAREN
prodExpr := powExpr (MUL powExpr | DIV powExpr)*
powExpr := unaryExpr | baseExpr
unaryExpr := SUB baseExpr | baseExpr
baseExpr := INTLIT | subscriptIdentifier | LPAREN expr RPAREN | forExpr
setDecl := LTHAN idGeneralDeclList GTHAN
idSubDeclList := (subscriptIdentifierDecl)+
idGeneralDeclList := (idDeclGeneral)+
idDeclList := (identifierDecl)+
idSubList := (subscriptIdentifier)+
idList := (identifier)+
idDeclGeneral := idDeclRange | subscriptIdentifierDecl
idDeclRange := identifierDecl LBRACKET expr COLON expr RBRACKET
subscriptIdentifierDecl := identifier (SUBSCRIPT (ID | INTLIT))?
identifierDecl := ID
subscriptIdentifier := identifier (SUBSCRIPT (ID | INTLIT))?
identifier := ID
ID := (’a’..’z’ | ’A’..’Z’)(’a’..’z’ | ’A’..’Z’ | ’0’..’9’)*
INTLIT := ’0’ | (’1’..’9’)(’0’..’9’)*
```